Load-Balanced Local Time Stepping for Large-Scale Wave Propagation

Max Rietmann, Daniel Peter, Olaf Schenk, Bora Uçar, Marcus J. Grote

Load-Balanced Local Time Stepping for Large-Scale Wave Propagation

Max Rietmann, Daniel Peter, Olaf Schenk
Institute for Computational Science
Universitá della Svizzera Italiana
Lugano, Switzerland
max.rietmann@usi.ch
olaf.schenk@usi.ch
peterd@usi.ch

Bora Uçar
CNRS and LIP (CNRS, ENS Lyon, UCBL, INRIA, Université de Lyon), France
bora.ucar@ens-lyon.fr

Marcus J. Grote
Institute of Mathematics
University of Basel, Switzerland
marcus.grote@unibas.ch

***Abstract*—In complex acoustic or elastic media, finite element meshes often require regions of refinement to honor external or internal topography, or small-scale features. These localized smaller elements create a bottleneck for explicit time-stepping schemes due to the Courant-Friedrichs-Lewy stability condition. Recently developed local time stepping (LTS) algorithms reduce the impact of these small elements by locally adapting the time-step size to the size of the element. The recursive, multi-level nature of our LTS scheme introduces an additional challenge, as standard partitioning schemes create a strong load imbalance across processors. We examine the use of multi-constraint graph and hypergraph partitioning tools to achieve effective, load-balanced parallelization. We implement LTS-Newmark in the seismology code SPECFEM3D and compare performance and scalability between different partitioning tools on CPU and GPU clusters using examples from computational seismology.**

I. Introduction

Efficiently simulating wave propagation at large scales has many important scientific and industrial application domains. In the field of seismology, simulating seismic waves resulting from an earthquake or other seismic source is an important modality used to better understand the Earth's interior structure and dynamic behavior. Many applications in both forward and inverse modeling have been pushing limits of traditional high-performance computing (HPC) resources for many years. Much of the optimization work in this field is focused on improving the implementation of standard algorithms, which can have bottlenecks that only better algorithm design can remove. Transformative improvements to simulation performance will likely require a coupling of algorithmic, hardware, and software improvements.

Although there are a handful of comparable spatial discretizations, higher-order finite elements have become a popular choice due to their flexibility and efficiency. The finite element mesh can adapt to the user's modeling requirements, conforming to external and internal topography, localized physics, and changing material properties. However, when an explicit time-stepping scheme is used, any particularly small elements (relative to the average element size) require small time steps for stability and thus create a time-stepping bottleneck.

Local time stepping (LTS) methods have been introduced to localize the time-step size to the element size, and can minimize the effect of these small elements on the overall performance [5]. These LTS methods, however, create a load-balancing problem when the simulation is run in parallel across many processors, a requirement for seismological applications. This paper is particularly focused on solving this load-balancing problem and presents several solutions using multi-constraint graph and hypergraph partitioning algorithms. We compare these algorithms for a variety of examples on both CPU and GPU clusters using our newly developed high-performance implementation.

This section further introduces the time-stepping bottleneck in detail. Section II introduces LTS and our explicit LTS-Newmark algorithm. Section III, the focus of the paper, introduces the load-balancing problem and detail our solutions. Section IV highlights performance experiments comparing the various solutions on several examples motivated by applications in seismology, followed by the conclusion.

A. Elastic Wave Equation

Although LTS is generally applicable for hyperbolic PDEs, we are specifically interested in the elastic wave equation, which models wave propagation through an elastic medium such as the Earth's crust and mantle. We currently ignore attenuation, which allows for frequency dependent damping of solutions, and leave it for future work.

For the displacement $\vec{u}$ with $x, y, \text{and } z$ components we have

$$\rho(\vec{x})\frac{\partial^2 \vec{u}}{\partial t^2} - \nabla \cdot \mathbf{T}(\vec{x},t) = f(\vec{x}_s,t)\ , \tag{1}$$

which is subject to a boundary condition with $\hat{\mathbf{r}} \cdot \mathbf{T} = 0$ on the free surface with outward normal $\hat{\mathbf{r}}$. At the vertical and lower boundaries we impose an absorbing boundary condition. The stress $\mathbf{T}(\vec{x},t)$ is related to the displacement gradient $\nabla\vec{u}$ via Hooke's constitutive law

$$\mathbf{T}(\vec{x},t) = \mathbf{C}(\vec{x}) : \nabla\vec{u}(\vec{x},t)\ , \tag{2}$$

where $\mathbf{C}$ is the fourth-order elasticity tensor with 21 independent parameters in the fully anisotropic case.

B. SEM Discretization

Spatial discretization is done via the so-called spectral element method (SEM), a special case of the standard continuous Galerkin finite element method. Restricted to hexahedra in three dimensions, the standard continuous finite element formulation [2], [13], leads to the following system

$$\mathbf{M}\ddot{\mathbf{u}} + \mathbf{K}\mathbf{u} = \mathbf{F}\,, \tag{3}$$

where $\mathbf{M}$ and $\mathbf{K}$ are called the mass and stiffness matrix, respectively. By using nodal Lagrange basis functions defined at the Gauss-Legendre-Lobatto (GLL) collocation points, Gauss quadrature yields a diagonal approximation of $\mathbf{M}$ that retains the high-order convergence properties. Thus, we can rewrite (3) in a form amenable to explicit time-stepping schemes (where $\mathbf{M}^{-1}$ is trivial to compute)

$$\ddot{\mathbf{u}} = -\mathbf{M}^{-1}\left(\mathbf{K}\mathbf{u} - \mathbf{F}\right). \tag{4}$$

C. Explicit Newmark Method

Although there are many choices of time-stepping schemes, we restrict ourselves to the explicit Newmark time-stepping scheme, a popular scheme in the wave-propagation community, due to its conservative nature. We apply the method to (4) with $\mathbf{u}(t)$ and $\mathbf{v}(t)$ steps staggered in time by $\Delta t/2$,

$$\mathbf{v}_{n+1/2} = \mathbf{v}_{n-1/2} - \Delta t\,\mathbf{M}^{-1}\mathbf{K}\mathbf{u}_n, \tag{5}$$

$$\mathbf{u}_{n+1} = \mathbf{u}_n + \Delta t\,\mathbf{v}_{n+1/2}, \tag{6}$$

where $\mathbf{u}_{n+\xi} = \mathbf{u}(t_n + \xi\Delta t)$. As with any explicit scheme, Newmark is only *conditionally* stable, requiring that the time step Δt meet the following Courant-Friedrichs-Lewy (CFL) condition, which limits the time-step size proportional to the element size normalized by the element's compressional wave velocity

$$\Delta t = C_{\text{CFL}} \min_{i\in\Omega_h}\left(\frac{h_i}{c_i}\right) \tag{7}$$

for a finite element mesh Ω_h, h_i is the characteristic element size of a mesh element, and c_i is the corresponding pressure velocity. Thus, Δt is determined by the globally smallest ratio (h_i/c_i), often found at the smallest element. This creates the peculiar situation where, e.g., a small element on a squeezed surface feature determines the time step for the entire mesh.

II. LTS Theory

Until the advent of LTS or multi-rate schemes, one could only avoid the CFL bottleneck by using implicit time-stepping schemes because they are usually unconditionally stable, and the choice of Δt depends only on the ability to resolve the desired solution. However, this requires solving a linear system involving $\mathbf{M}$ and $\mathbf{K}$, which is much more difficult on a distributed memory system than the simple synchronizations at partition boundaries required by any explicit scheme.

Previously, Dumbser, Käser, and Toro. [6] have shown the effectiveness of LTS in their ADER-DG scheme, which allows the time step to be adapted to each element uniquely, but is limited to a discontinuous Galerkin (DG) implementation. Similarly, Gödel et al. [7] have proposed a two-level multi-rate scheme for Maxwell's equations using GPUs.

We base our work closely on the original work done by Diaz and Grote [5], who derive an LTS scheme for the leap–frog time-stepping scheme, which has been implemented using DG in [11]. Leap–frog is equivalent to an explicit Newmark scheme, thus our two-level LTS-Newmark is also identical to their LTS-leap–frog. Following [5] we split the degrees of freedom using a selection matrix $\mathbf{P}$, and thus the mesh into *fine* $\mathbf{P}$ and *coarse* $(\mathbf{I}-\mathbf{P})$ regions

$$\mathbf{u}(t) = \mathbf{P}\mathbf{u}(t) + (\mathbf{I}-\mathbf{P})\mathbf{u}(t), \tag{8}$$

where $\mathbf{P}$ has 1 on the diagonal corresponding to a fine node, and 0 elsewhere. The fine region takes steps of size $\Delta t/p$ ($p \in \mathbb{N}$) and the coarse region takes steps of size Δt, such that the CFL condition (7) is satisfied everywhere.

If the mesh in question has a relatively small number of small elements, we will save a lot of work by stepping most of elements p times less. We model this performance speedup simply by

$$\text{speedup model} = \frac{p \times \#\text{ elements}}{p \times \#\text{ fine elements} + \#\text{ coarse elements}}. \tag{9}$$

As the number of coarse elements reaches the total number of elements, we see that the speedup approaches p.

This previous LTS work [5], however, limits itself to only two regions, fine and coarse. Building on these two-level results, we propose a scheme that allows for multiple time-stepping levels for greater flexibility and greater performance. We also present details on the high-performance implementation of this multi-level scheme. Grote et al. has proposed several LTS schemes including leap–frog [5], Adams–Bashforth, and Runge–Kutta. Given the popularity of explicit Newmark in the wave-propagation community, we decided to derive an LTS scheme for Newmark as well. However, the parallelization strategy presented in this paper is relatively general and useful to any LTS scheme.

A. Newmark LTS

We begin by making the following approximation,

$$\mathbf{A}\mathbf{P}\mathbf{u}(t) + \mathbf{A}(\mathbf{I}-\mathbf{P})\mathbf{u}(t) \approx \mathbf{A}\mathbf{P}\tilde{\mathbf{u}}(\tau) + \mathbf{A}(\mathbf{I}-\mathbf{P})\mathbf{u}(t_n), \tag{10}$$

where $\mathbf{A} = \mathbf{M}^{-1}\mathbf{K}$ and our coarse wave field $(\mathbf{I}-\mathbf{P})\mathbf{u}(t)$ is approximated by the constant evaluated at the current time t_n. The fine part $\mathbf{P}\mathbf{u}(t)$ is approximated by $\mathbf{P}\tilde{\mathbf{u}}(\tau)$, which is a solution to the following approximate first-order formulation of (4):

$$\begin{aligned} \frac{d\tilde{\mathbf{u}}}{d\tau}(\tau) &= \tilde{\mathbf{v}}, \\ \frac{d\tilde{\mathbf{v}}}{d\tau}(\tau) &= -\mathbf{A}\mathbf{P}\tilde{\mathbf{u}}(\tau) - \mathbf{A}(\mathbf{I}-\mathbf{P})\mathbf{u}(t_n) \end{aligned} \tag{11}$$

with initial conditions $\tilde{\mathbf{u}}(0) = \mathbf{u}(t_n)$ and $\tilde{\mathbf{v}}(0) = 0$. Note that (11) is time reversible, that is $\tilde{\mathbf{u}}(\tau) = \tilde{\mathbf{u}}(-\tau)$ and $\tilde{\mathbf{v}}(\tau) = -\tilde{\mathbf{v}}(-\tau)$.

To derive the LTS-Newmark scheme, we begin by deriving Newmark using an integral formulation. By splitting the system of differential equations (4) into our newly partitioned

fine and coarse variable approximations and integrating the resulting expression, we obtain

$$\begin{aligned} \mathbf{v}(t_n + \Delta t/2) &= \mathbf{v}(t_n - \Delta t/2) - \Delta t \mathbf{A}(\mathbf{I} - \mathbf{P})\mathbf{u}(t_n) \\ &- \int_{-\Delta t/2}^{\Delta t/2} \mathbf{A}\mathbf{P}\tilde{\mathbf{u}}(s)\, ds, \\ \mathbf{u}(t_n + \Delta t) &= \mathbf{u}(t_n) + \Delta t\, \mathbf{v}(t_n + \Delta t/2)\ . \end{aligned} \tag{12}$$

Following the same procedure for (11), we derive similarly

$$\begin{aligned} \tilde{\mathbf{v}}(\Delta t/2) &= \tilde{\mathbf{v}}(\Delta t/2) - \Delta t \mathbf{A}(\mathbf{I} - \mathbf{P})\mathbf{u}(t_n) \\ &- \int_{-\Delta t/2}^{\Delta t/2} \mathbf{A}\mathbf{P}\tilde{\mathbf{u}}(s)\, ds, \\ \tilde{\mathbf{u}}(\Delta t) &= \mathbf{u}(t_n) + \Delta t \tilde{\mathbf{v}}(\Delta t/2)\ . \end{aligned} \tag{13}$$

Combining (12) and (13) and solving for $\mathbf{v}(t_n + \Delta t/2)$ yields the following equations, nearly completing our LTS-Newmark scheme:

$$\begin{aligned} \mathbf{v}_{n+1/2} &= 2\left(\frac{\tilde{\mathbf{u}}(\Delta t) - \mathbf{u}(t_n)}{\Delta t}\right) + \mathbf{v}(t - \Delta t/2), \\ \mathbf{u}_{n+1} &= \mathbf{u}_n + \Delta t \mathbf{v}_{n+1/2}\ . \end{aligned} \tag{14}$$

However, this two-level restriction limits the total efficiency of an LTS algorithm. Thus, we enhance the method to include additional levels, in a recursive manner.

B. Multiple levels

We begin by defining a series of nonoverlapping selection matrices, $\mathbf{P}_k$,

$$\sum_{}^{N} \mathbf{P}_k = \mathbf{I}, \quad \mathbf{P}_j \mathbf{P}_k = 0, j \neq k\ . \tag{15}$$

Here $\mathbf{P}_1$ represents nodes with elements in the *coarsest* level with time-step Δt, whereas $\mathbf{P}_N$ is the finest level with step size $\Delta t/p_N$, $p_N \geq p_{1<k<N}$. We further require that bordering elements in different levels take steps that are a multiple (two $\Delta t/4$ fit into a $\Delta t/2$). For simplicity, we limit ourselves to powers of two for p,

$$\Delta t/p_k, \quad p_k = 2^{k-1}, \qquad k = 1, 2, \ldots, N\ , \tag{16}$$

in order to meet this constraint.

Our concurrent paper [15] details the theoretical results for both the two-level and multi-level LTS-Newmark algorithms, where the two-level (via the LTS-leap–frog scheme [5]) and multi-level Newmark schemes are shown to preserve convergence and conservation properties.

For simplicity, we derive a three-level scheme, which simply embeds the additional variable $\hat{\mathbf{u}}(s)$ into the original two-level system from (11):

$$\begin{aligned} \frac{d\tilde{\mathbf{u}}}{d\tau}(\tau) &= \tilde{\mathbf{v}}, \\ \frac{d\tilde{\mathbf{v}}}{d\tau}(\tau) &= -\mathbf{A}\mathbf{P}_1\mathbf{u}(t) - \mathbf{A}\mathbf{P}_2\tilde{\mathbf{u}}(\tau) - \mathbf{A}\mathbf{P}_3\hat{\mathbf{u}}(s), \end{aligned} \tag{17}$$

where $\hat{\mathbf{u}}(s)$ satisfies the additional system

$$\begin{aligned} \frac{d\hat{\mathbf{u}}}{ds}(s) &= \hat{\mathbf{v}}, \\ \frac{d\hat{\mathbf{v}}}{ds}(s) &= -\mathbf{A}\mathbf{P}_1\mathbf{u}(t) - \mathbf{A}\mathbf{P}_2\tilde{\mathbf{u}}(\tau) - \mathbf{A}\mathbf{P}_3\hat{\mathbf{u}}(s) \end{aligned} \tag{18}$$

with initial conditions $\hat{\mathbf{u}}(0) = \tilde{\mathbf{u}}(\tau)$ and $\hat{\mathbf{v}}(0) = 0$ and noting that both $\mathbf{u}(t)$ and $\tilde{\mathbf{u}}(\tau)$ are constant. Following a technique similar to the derivation of the two-level system, we present a three-level scheme shown in Algorithm 1.

Algorithm 1 LTS-Newmark three-level

Require: $\mathbf{u}_0, \mathbf{v}_{-1/2}, \Delta\tau = \Delta t/p_2, \Delta s = \Delta t/p_3$
  $\tilde{\mathbf{u}}_0 = \mathbf{u}_0, \tilde{\mathbf{v}}_0 = 0$
  for $n = 0, \ldots, T_n$ **do**
    $\tilde{\mathbf{u}}_0 = \mathbf{u}_n$
    $\mathbf{w} = \mathbf{A}\mathbf{P}_1\mathbf{u}_n$
    for $m = 0, \ldots, (p-1)$ **do**
      $\hat{\mathbf{u}}_0 = \tilde{\mathbf{u}}_m$
      $\mathbf{z} = \mathbf{A}\mathbf{P}_2\tilde{\mathbf{u}}_m$
      $\hat{\mathbf{v}}_{1/2} = -\Delta s/2\,(\mathbf{w} + \mathbf{z} + \mathbf{A}\mathbf{P}_3\hat{\mathbf{u}}_0)$
      $\hat{\mathbf{u}}_1 = \hat{\mathbf{u}}_0 + \Delta s\, \hat{\mathbf{v}}_{1/2}$
      for $s = 1, \ldots, (p_2/p_3 - 1)$ **do**
        $\hat{\mathbf{v}}_{s+1/2} = \hat{\mathbf{v}}_{s-1/2} - \Delta s\,(\mathbf{w} + \mathbf{z} + \mathbf{A}\mathbf{P}_3\hat{\mathbf{u}}_s)$
        $\hat{\mathbf{u}}_{s+1} = \hat{\mathbf{u}}_s + \Delta s\, \hat{\mathbf{v}}_{s+1/2}$
      end for
      $m = 0 : \tilde{\mathbf{v}}_{1/2} = (\hat{\mathbf{u}}_{p_2/p_3} - \tilde{\mathbf{u}}_m)/\Delta\tau$
      $m > 0 : \tilde{\mathbf{v}}_{m+1/2} = \tilde{\mathbf{v}}_{m-1/2} + 2(\hat{\mathbf{u}}_{p_2/p_3} - \tilde{\mathbf{u}}_m)/\Delta\tau$
      $\tilde{\mathbf{u}}_{m+1} = \tilde{\mathbf{u}}_m + \Delta\tau\tilde{\mathbf{v}}_{m+1/2}$
    end for
    $\mathbf{v}_{m+1/2} = \mathbf{v}_{m-1/2} + 2(\tilde{\mathbf{u}}_{p_2} - \mathbf{u}_n)/\Delta t$
    $\mathbf{u}_{n+1} = \mathbf{u}_n + \Delta t \mathbf{v}_{m+1/2}$
  end for

From this three-level scheme, the generalization to many levels is accomplished by recursively embedding additional levels $\mathbf{P}_{k+1}$ into each $\mathbf{P}_k$. Thus, before the $\Delta t/p_k$ step can complete, each level below must complete their respective steps.

C. Implementation for a SEM

The several existing high-performance implementations of LTS mentioned previously [6], [7], [11] all utilize DG discretization schemes that are more expensive than a SEM for a fixed mesh, but are more amenable to an efficient LTS implementation due to the duplicated degrees of freedom at the discontinuous element boundaries. The action of $\mathbf{A}\mathbf{P}\tilde{\mathbf{u}}$ only contributes to nodes in $\mathbf{P}$, and the coupling between coarse and fine is done via the numerical flux, greatly simplifying the implementation. The finer-level vector updates (e.g., $\hat{\mathbf{u}}_{s+1} = \hat{\mathbf{u}}_s + \Delta s\, \hat{\mathbf{v}}_{s+1/2}$) are restricted to the active degrees of freedom in $\mathbf{P}$.

In contrast to DG, a SEM shares nodes between elements, meaning that $\mathbf{A}\mathbf{P}\tilde{\mathbf{u}}$ contributes to both nodes in $\mathbf{P}$ and $\mathbf{I} - \mathbf{P}$. Likewise, $\mathbf{A}(\mathbf{I} - \mathbf{P})\mathbf{u}$ contributes to nodes in both $\mathbf{P}$ and $\mathbf{I} - \mathbf{P}$ as well. The mixing of information between levels happens exclusively at elements which share $\mathbf{P}$ and $(\mathbf{I} - \mathbf{P})$ nodes, and thus requires special care in the implementation. In the previous two- and three-level algorithms, the work-saving feature of LTS is implicit, and working out the minimal set of required numerical operations for a high-performance version of the code requires great care.

Our choice of SEM and Newmark, and thus the implementation basis of LTS-Newmark, is motivated by the popular computational seismology package SPECFEM3D *Cartesian* [13], which can input user-defined hexahedral meshes

and is commonly run using fourth-order elements with 125 nodes per element. Known to perform well at large scale, a GPU version was added in 2012 [14], which we now extend with LTS for both CPUs and GPUs. Our LTS contributions to the code are being made available as part of the open source package hosted on GitHub, allowing interested developers to see details of the implementation in code.

The progressively smaller levels of LTS are easily modeled recursively, which we mirror in the code using recursive calls of an `lts_global_step()` routine, called from the global time-stepping routine. By ensuring that all operations specific to LTS are optimized to perform only necessary operations and memory transfers, we can achieve a single-threaded efficiency of greater than 90%, relative to an ideal speedup modeled in Eq. (9). However, we also require an efficient parallel implementation, which we address in the next section.

III. The Partitioning Problem

Real-world seismology problems are too big for a single machine in both memory and compute cost and thus require parallelization. Packages such as SPECFEM3D follow the traditional parallelization approach, where a finite element mesh is partitioned using a tool such as SCOTCH [12] or MeTiS [9]. In the standard approach, the stiffness matrix contribution is computed for partition boundaries first, which are then exchanged using asynchronous MPI communications. We follow this approach in the GPU implementation: the asynchronous overlapping is repeated for the required GPU-CPU memory copies as well.

Due to LTS, however, some elements take more steps and thus require more work than other elements. An unmodified partitioner will lead to a parallelization that is unbalanced, reducing overall performance. Standard partitioning packages such as SCOTCH allow weighted elements and produce partitions that are weight balanced according to these inputs. Unfortunately, this weighting technique also has a balancing problem, due to the way an LTS scheme steps through the elements in time.

Indeed LTS introduces an additional balancing constraint to the partitioning of the mesh. As seen in the multi-level algorithm, the steps of the LTS algorithm proceed recursively through the lower levels, until finally taking a step on the coarsest global level. We define an *LTS cycle* as the work needed to take all steps at every level until the coarsest level takes a step of size Δt.

For instance, Fig. 1 shows a one-dimensional time-stepping diagram with two partitions from a standard partitioner, and the corresponding unbalanced timeline. Each of the four fine-level steps requires synchronization between the partitions. Furthermore, since the partition A has three times more fine elements than partition B, processor A will take three times longer than processor B to complete a single fine-level step $\Delta\tau$. Once the fine level (Ω_f) completes, processor A will stall waiting for processor B due to the imbalance of the coarse elements. This imbalance on each refinement level across the two processors will drastically reduce the speedup achieved by the single-core LTS implementation. Furthermore, the parallelization should additionally consider the increased communication requirements between elements with $p > 1$.

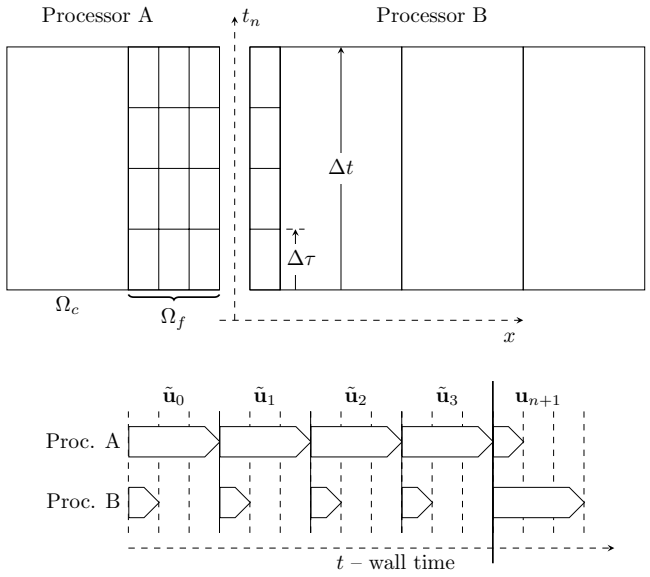

Fig. 1. A timeline of a 1D mesh with two partitions that are balanced without consideration for LTS. The partition for processor A has three fine elements, and a single coarse element, whereas partition B has only a single fine element and three coarse elements. The top graphic shows how many steps per Δt each element must take, and the bottom graphic depicts a runtime profile showing the fine-level time steps $\tilde{\mathbf{u}}_m$ and the required synchronization between partitions at every step. We note that processor B stalls waiting for processor A (and vice versa) due to the coarse-fine imbalance.

A scalable parallelization solution is thus critical to the success of LTS in real-world applications. For the scope of this paper, we present a solution relying on partitioning tools that provides good scalability and can be easily adopted into existing code bases that use traditional partitioning techniques. We call this solution *p-level balanced partitioning*, as it attempts to balance the load across each level of refinement (p-level) using existing partitioning tools by partitioning each p-level equally across processors.

The previously mentioned LTS implementations did not ignore this parallelization problem. In the two-level scheme proposed by [7], in order to allow for multiple GPUs, the partitioning is restricted to only cut across coarse ($p = 1$) elements, ensuring that MPI synchronization is only required every Δt and not for any substeps. Weighted elements via MeTiS ensure good load balancing. We also considered this approach, but rejected it because it inherently limits the scalability with an artificially high lower limit on the number of elements per partition. At some point the partition cannot be split further without cutting across fine-level ($p > 1$) elements.

In the multi-level ADER-DG scheme [6], elements of a similar time-step size are grouped (analogous to our p-levels) and partitioned individually and re-merged with the other groups to provide a single partition per processor. This is very similar to our SCOTCH-P approach to be introduced in Sec. III-B and motivated the use of a multi-constraint partitioner to do this in a single step without the need for re-merging partitions.

With our initial partitioning goals set, we can now discuss how LTS further changes the partitioning cost, and compare graph and hypergraph partitioning approaches, including the

communication and multi-constraint modeling for each type. Then, we introduce several implementations using a variety of partitioning libraries (compared later in Section IV).

A. LTS-Partition models: Graphs and hypergraphs

To design an LTS-aware partitioning algorithm, we need to model the LTS requirements in terms of load balance and communication costs. Ensuring that each processor is equally loaded for each $\Delta t/p$ substep is vital for parallel efficiency of the implementation. Existing graph partitioning tools can balance work between partitions by weighting the graph vertices, which can be used to balance cheaper acoustic domains with more expensive elastic ones, for example. However, as noted, the recursive nature of LTS requires additional balancing inputs, one for each level.

The standard graph and hypergraph partitioning problems (which are NP-complete) ask for a partition of the vertices of the given (hyper)graph in a given number K of nonempty, disjoint sets. The partitioning constraint of both problems is to achieve a balance on the part weights, usually defined as the total weight of the constituting vertices. The partitioning objective, called cutsize, is to reduce the number or weight of the edges having vertices in different parts in the graph partitioning problem. In the hypergraph partitioning problem a function of the hyperedges straddling the partition boundaries is the objective function.

A recent variant of the standard graph and hypergraph partitioning problem blends multiple balance constraints, which is therefore called the *multi-constraint graph and hypergraph partitioning problem* [1], [3], [8]. In this problem, each vertex has a vector of weights. We use $w[v, i]$ to denote the P weights associated with the vertex v, for $i = 1, \ldots, P$. For a vertex set U, we use $W[U, i]$ to denote the sum of the ith weights of vertices in U, i.e., $W[U, i] = \sum_{u \in U} w[u, i]$. In this setting, the partitioning constraint is to satisfy a balance criterion for each $i = 1, \ldots, P$:

$$W[V_k, i] \leq (1 + \varepsilon)\frac{W[V, i]}{K}, \text{for } k = 1, \ldots, K \qquad (19)$$

for an allowed imbalance ε (the partitioning objective remains the same for both the of two partitioning problems). Besides requiring more computational work, elements in a finer level (e.g., $\Delta t/2$) also create p-times more communication volume when split between processors. This higher cost is visualized in Fig. 2, an example (higher-order) two-dimensional finite element mesh with 9 nodes per element. The black nodes are in a higher p-level ($p = 2$) and thus require more communication when cut. The bordering gray nodes are a type of "halo" due to the LTS algorithm for continuous elements, and also require updates on each $\Delta t/2$ step. Visualized in each of the lower meshes, the communication cost associated with the three possible partition cuts is shown to highlight how LTS changes the cost associated with the cuts along the different levels of the mesh. In order to partition our mesh across processors, we turn to graph and hypergraph partitioning tools.

1) Graphs: A mesh itself is a bipartite graph [12], where elements are defined by their corner vertices (nodes), and vertices are connected to multiple elements. For a standard graph partitioning tool such as SCOTCH or MeTiS, we first create the mesh's dual graph, which represents the connection

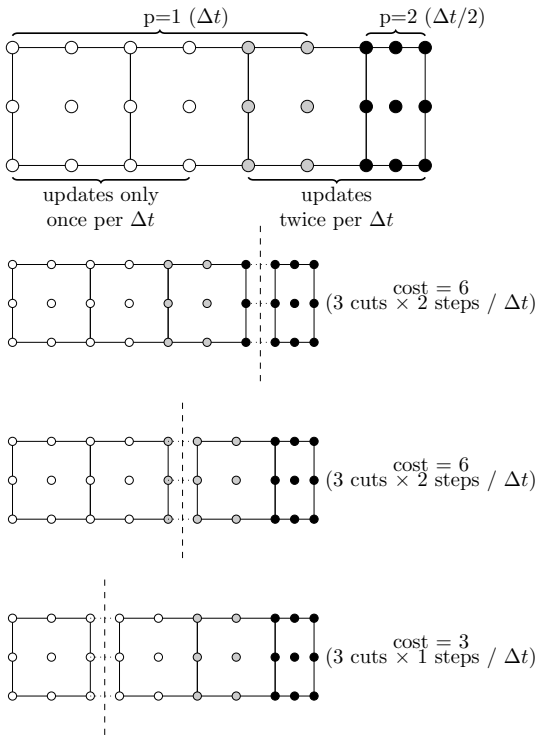

Fig. 2. A 2D finite element mesh depicting the partitioning cost for various cuts. The nodes in black and the nodes in gray are updated at every $\Delta t/2$ step. A cut across black or gray nodes will require 2 synchronizations for every LTS cycle (Δt step).

of mesh elements across faces as seen in Fig. 3. The dual graph only accounts for the communication requirements of nodes on an element's face. Corner nodes, on the other hand, are connected to multiple elements and are not modeled by this simply connected graph.

In the absence of LTS, vertices and edges are generally unweighted. For tools that support this, correct load balancing for LTS is only achievable through the multi-constraint approach previously mentioned. Each vertex is assigned the weight vector $w[v, i]$ corresponding to the load of the associated element at level i, which should be balanced by the partitioning library appropriately.

The edges of the graph are also weighted according to the p-level, with the weight of the edge set to the maximum value of p for the two connected vertices. This edge weighting can only approximate the cost detailed in Fig. 2. To build a fully accurate cost model, we require a hypergraph, which allows the edges to connect all relevant vertices.

2) Hypergraphs: Here we first formally describe the standard hypergraph model for the (typical) finite element mesh and the associated partitioning problem, to set the scene for an accurate hypergraph model for LTS. Recall that a hypergraph is an ordered pair $H = (V, N)$ consisting of a set V of vertices and a set N of hyperedges (or nets), where each hyperedge is a subset of the vertex set V, and that the vertices can have weights and the hyperedges can have costs. We again use $w[v]$ and $w[U] = \sum_{u \in U} w[u]$ to denote the weight of a vertex v and the sum of the weights of vertices in the vertex set U, respectively. We use $c[h]$ to denote the cost of a hyperedge h.

In the hypergraph model of a mesh, the vertices correspond to elements, and the hyperedges correspond to the corner nodes. Each corner node defines a hyperedge and connects all elements (vertices) touching it. Thus a corner node might connect 4 or more elements in two dimensions, and 8 or more elements in three dimensions. A simple 2D example is shown in Fig. 3. The diagram shows a 4-element rectangular mesh and its corresponding dual graph and hypergraph. In the case where all 4 elements are in a separate partition, the dual graph will only count 4 edges in cut, where the hypergraph will add the additional cuts between all 4 elements due to the central node, modeling the additional required MPI communication accurately.

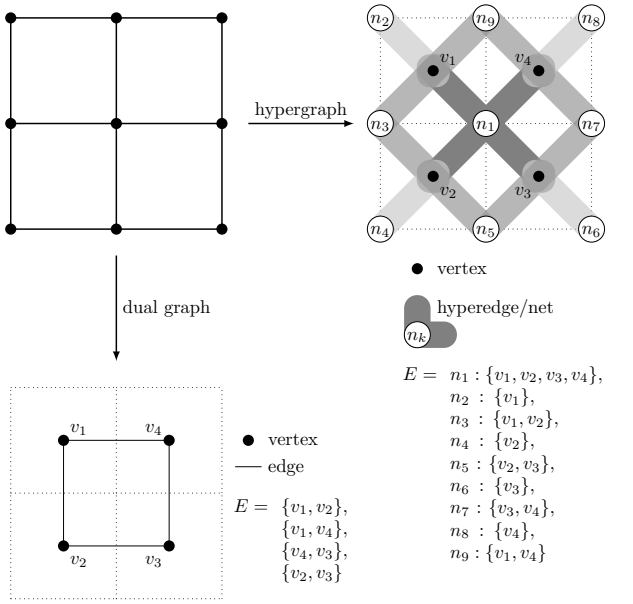

Fig. 3. Graph vs. hypergraph representations of a 2D finite element mesh. The traditional dual-graph can only model the connection between elements sharing a face, where the hypergraph models the connection between all elements that share a node (e.g., when four elements share a corner).

$\Pi = \{V_1, \ldots, V_K\}$ is a K-way vertex partition of a given hypergraph $H = (V, N)$, if each part is a nonempty subset of V, parts are pairwise disjoint (that is, $V_i \cap V_j = \emptyset$ for $i \neq j$), and collectively exhaustive (that is, $V = \bigcup V_i$). A K-way vertex partition Π is *balanced* if (19) holds for $V_1, \ldots, V_K$ with $P = 1$. The cost of a vertex partition Π of a given hypergraph $H = (V, N)$ is called the *cut size*. It measures the degree of the spread of the hyperedges that have vertices in different parts, weighted according to the cost of the hyperedges. There are various cut size definitions [10]. The adequate one for our purposes is the following:

$$\text{cutsize}(\Pi) = \sum_{h \in N} c[h](\lambda_h - 1), \tag{20}$$

where λ_h is the number of parts in which the hyperedge h has vertices.

Given a hypergraph $H = (V, N)$, an integer K, and an imbalance parameter ε, the hypergraph partitioning (HP) problem asks for a balanced (19), K-way vertex partition Π, with the minimum cut size (20).

We now propose a hypergraph model for partitioning meshes for load balanced LTS computations with reduced communication cost. For a given finite element mesh, we create a hypergraph $H = (V, N)$ in two steps. In the first step, we define vertices and their weights so that a balanced partitioning of the vertices will correspond to a balanced computational load distribution among processors at all LTS levels. In the second step, we define hyperedges and their costs such that the total volume of communication in an LTS cycle will exactly match the cut size (20), when the elements are partitioned according to the vertex partitions.

The vertices and their weights in H are defined as follows. Each element of the mesh is uniquely represented by a vertex in V. As an element belongs to a level, the corresponding vertex in H can be said to belong to a level. We associate a weight vector of size P, where P is the number of levels, with each vertex. In this setting, the vertex weight vector $w[v, i]$ is set to one for i corresponding to the level of the associated element, and all other weight coordinates are set to zero. Once these are set, we partition the vertices of H into K parts, where the balance criteria (19) are satisfied for all weight coordinates. After assigning each part to a processor, we obtain a (hopefully) load balanced partitioning such that the computational work is evenly distributed among processors across all LTS levels.

The hyperedges and their costs in H are defined as follows. For each node n of the mesh, we create a hyperedge h_n, where h_n contains vertices corresponding to the elements containing the node n. We put a copy of this hyperedge to the hypergraph for each element containing the node n. We use $h_n^{(1)}, \ldots, h_n^{(e)}$ to refer to the e copies of the hyperedge, where $e = |elmnts(n)|$ is the number of elements containing n. Thus, when the set of elements $elmnts(n)$ is assigned to λ_n different processors, for each element in $elmnts(n)$, $\lambda_n - 1$ messages need to be delivered from the owner of the element to other processors. Now, as each element belongs to a particular LTS level, the messages should be delivered according to the step size of the level. Therefore, for each hyperedge, $c[h_n^{(i)}]$ is set to χ where χ is the level of the ith element in the set $elmnts(n)$. With this hyperedge and cost definitions, $\sum_{i=1}^{e} c[h_n^{(i)}](\lambda_n - 1)$ cost is added to the cut size, when the e elements containing the node n are partitioned among $\lambda_n - 1$ processors. Since the total volume of communication can be computed by the sum of the volume of communication per element, the cut size (20) represents the total volume of communication in an LTS cycle accurately. A simplification is possible here. Since all hyperedges $h_n^{(1)}, \ldots, h_n^{(e)}$ associated with the node n have the same set of vertices, we can represent them with a single hyperedge h'_n with $c[h'_n] = \sum_{i=1}^{e} c[h_n^{(i)}]$. Then, the number of hyperedges is reduced to the number of nodes in the mesh without any loss in the correspondence between the cut size and the total volume of communication.

B. Partitioning algorithms for LTS

Once the graph and hypergraph models are defined, we can develop methods to partition the mesh based on those models. We examined the following four techniques:

a) SCOTCH: This is the standard graph partitioner which is used in SPECFEM3D. It performs a standard parti-

tioning, but assigns a single weight to each element according to the p-level, such that each partition will have equal work (measured over a global Δt step), but will be unbalanced for substeps taken at different LTS levels. This provides a baseline to measure the relative success of a multi-constraint setup.

b) SCOTCH-P: SCOTCH itself provides only single-constraint partitioning. We propose the following approach to use SCOTCH beyond the baseline. Each p-level is partitioned separately among all processors using the standard SCOTCH routines, so that the partitions at all levels have a balanced load. We then map exactly one partition from each level to a single processor so that the processors have balanced load across all levels. While mapping partitions from each level, we greedily couple each partition from level 1 to the best available partition from level 2, and so on. One could experiment with more efficient mapping methods (based on weighted graph matchings), but we reserve this for future work.

c) MeTiS: As of version 5.0, MeTiS can perform a multi-constraint graph partition with weighted edges, attempting to balance p-levels and reduce the edge cut as an upper bound to the total communication volume simultaneously.

d) PaToH: It is a hypergraph partitioner [4] which performs a multi-constraint partitioning with weighted hyperedges to accurately minimize the total communication volume.

IV. Performance Experiments

As noted, we integrated the LTS-Newmark algorithm into the seismology software package SPECFEM3D. Primarily written in Fortran95, the CPU version remains a purely MPI-based code, such that a typical simulation is run using a single MPI rank per processor. The GPU version [14] extends the original code, wrapping calls to CUDA-C in order to launch the appropriate GPU kernels. The GPU version typically runs a single MPI rank per GPU, which is commonly 1 or 2 GPUs per supercomputing node.

A. Application Mesh Benchmarks

In order to test our LTS implementation at large scale, we assembled four test benchmark hexahedral meshes that replicate refinement seen in real-world applications. In Fig. 4 we show smaller examples of these benchmarks, with colored p-levels. The *trench* mesh is designed to model a long strip of refinement, a common problem seen in several application meshes, especially where two internal topographies meet and produce a long row of pinched elements. The *embedding* mesh is the simplest possible example of refinement and models any localized small-scale feature. The *crust* example models topography and large-scale surface features. Each of these benchmarks can, in principle, be scaled to any size and any level of LTS speedup. However, the crust mesh or other examples with topography, for instance, are limited in LTS speedup because of the large number of small elements on the surface, making it impossible to increase the ratio of large to small elements without making an unrealistically tall and skinny mesh. In this case, tetrahedral elements are better able to conform to a desired topography and body element size, and thus can yield a higher expected LTS speedup. However, as previously noted, we are currently limited to hexahedral elements in the absence of a stable and efficient quadrature rule for tetrahedra to allow for a diagonal mass matrix.

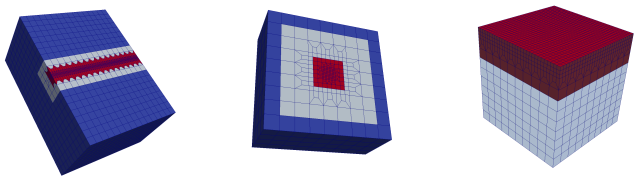

Fig. 4. Small examples of the four benchmark meshes used to compare and test the performance of each partitioning scheme along with LTS implementation performance. Actual performance benchmarks were conducted on larger examples of these meshes. Smallest p-level elements colored in red, mid-sized in gray, and largest in blue.

B. Partitioning Results

In order to compare each partitioning implementation, we conducted load balance, communication cost, and application performance experiments using large versions of each type of mesh depicted in Fig. 4. In Fig. 5 we see the size and theoretical speedup (9) of each mesh used for testing.

Mesh	# elements	# DOF	Theor. LTS speedup	# of levels
Trench	2.5M	170M	6.7	4
Trench Big	26M	1.7B	21.7	6
Embedding	1.2M	78M	7.9	4
Crust	2.9M	190M	1.9	2

Fig. 5. Benchmark meshes in detail. The fourth-order elements have 125 nodes per element, increasing the total degrees of freedom (DOF) by almost two orders of magnitude relative to the number of elements.

Figure 6 visualizes each partitioning implementation on the trench mesh.

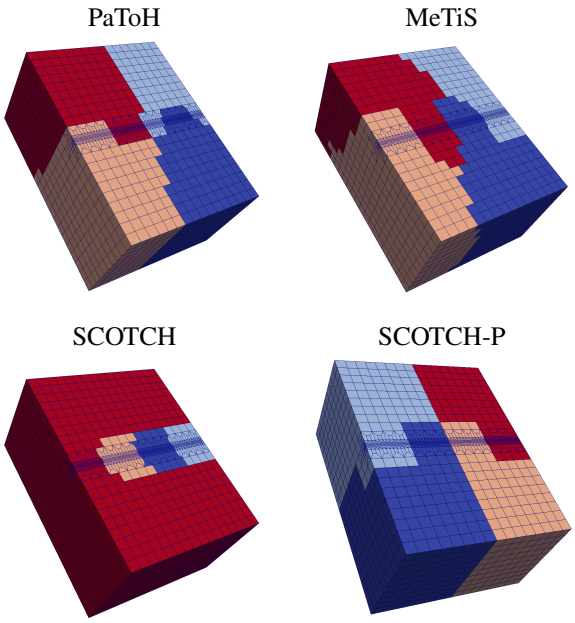

Fig. 6. All partitioning tools on example trench mesh with 4 partitions (partition seen by color). Note that SCOTCH (incorrectly) only balances work per LTS cycle, where the others balance each level correctly.

Obviously, the most important goal of each partitioner is to produce the highest possible application performance.

However, it is useful to compare partitioning performance in terms of graph cut, total communication volume, and load balance to help explain and understand application performance differences. For example, the PaToH partitioning library exposes many parameters, including the desired final imbalance (`final_imbal`), which affects the trade-off between communication cost (cuts along hyperedges) and the balance between p-levels across partitions.

We are interested in two metrics, starting with load imbalance defined as

$$\text{load imbalance}\,\% = \frac{(\text{max load}) - (\text{min load})}{(\text{max load})} \times 100, \quad (21)$$

where load is the estimated computational load per partition. We define load as the sum of graph vertices each weighted by their respective p-level refinement p (each element has approximate computational cost p). This is measured for both the entire mesh, and across p-levels.

Using the 2.5M element trench mesh, we compare the load imbalance across the MeTiS, PaToH, and SCOTCH-P partitioners in the table from Fig. 7. The imbalance table highlights several important points. The MeTiS multi-constraint partitioner is currently not able to maintain an optimal balance across levels, where the PaToH partitioner does manage this balance, where the `final_imbal` parameter can be used to improve the balance at the cost of additional communications.

	Load imbalance			
# of parts	MeTiS	PaToH 0.05	PaToH 0.01	SCOTCH-P
16	34%	11%	2%	6%
32	88%	17%	5%	6%
64	89%	19%	7%	7%

Fig. 7. Total work-load imbalance (21) for MeTiS, PaToH, and SCOTCH-P partitioners on 2.5M mesh. PaToH is additionally compared with two values of parameter `final_imbal=0.05,0.01`.

The second critical metric is the weighted graph cut and the total mpi-communications volume. The traditional partitioners MeTiS and SCOTCH-P both utilize weighted graph cut metrics, as opposed to PaToH, which optimizes the weighted hypergraph cut, which accurately models total communications volume as noted in Fig. 3. Again using the 2.5M element trench mesh, we compare graph cut and total communication volume metrics across each partitioner in Fig. 8.

	MeTiS		PaToH 0.05	
# of parts	Graph cut	MPI volume	Graph cut	MPI volume
16	1.4×10^6	1.0×10^7	1.8×10^6	1.1×10^7
32	2.4×10^6	2.0×10^7	2.9×10^6	1.8×10^7
64	3.5×10^6	3.0×10^7	4.2×10^6	2.6×10^7
	SCOTCH-P		PaToH 0.01	
16	1.9×10^6	1.3×10^7	1.0×10^6	1.0×10^7
32	3.1×10^6	2.1×10^7	2.3×10^6	1.6×10^7
64	4.7×10^6	3.3×10^7	3.4×10^6	2.3×10^7

Fig. 8. Table comparing communication cost metrics between MeTiS, PaToH (with `final_imbal=0.05,0.01`), and SCOTCH-P for refinement trench mesh with 2.5M elements. *Graph cut* is the weighted cost of cut edges for the simple graph, and *MPI volume* is the total MPI-communications volume per LTS cycle.

Although MeTiS is able to produce a better *graph cut*, the *MPI volume* is better optimized by PaToH and its more accurate hypergraph representation. However, as noted, when we examine load imbalance, MeTiS does not compare favorably.

We note that SCOTCH-P is able to beat both MeTiS and PaToH in terms of communication costs while maintaining a better load balance. The simple greedy reorganization used by SCOTCH-P after the p-level partitioning seems to work extremely well for the mesh examples we tested. From these partitioning experiments, we expect SCOTCH-P and PaToH to perform well in the application performance experiments in the next section, where we also can evaluate the effect of the load imbalance and communication cost trade-off for PaToH.

C. CPU and GPU Performance results

We ran benchmarks on the large CPU and GPU cluster *Piz Daint*. Each compute node is powered by a single 8-core Intel E5-2670, and a single NVIDIA Tesla K20X, where the CPU version runs 1 process per core (8 per node) and the GPU version runs 1 process per GPU (1 per node), and we compare performance on a node-to-node basis. Because LTS improves the efficiency of the time-stepping method, we must evaluate performance in terms of the wall-clock time to compute a fixed amount of simulation time. More simply, we are interested in the wall-clock time (in seconds) it takes to simulate, e.g., $T = 100$ seconds of wave propagation. A non-LTS scheme is forced to take the globally smallest time step ($\Delta t_{\min} = \Delta t / p_{max}$), and we measure the time it takes to simulate $T/(\Delta t_{\min})$ steps. LTS, on the other hand, takes steps of different sizes, globally synchronized every Δt, such that every Δt of simulated time is less expensive compared to the non-LTS scheme. Thus performance is measured as [simulated time]/[wall clock time] (s/s), however we opt instead to present our results normalized (relative) to the non-LTS (reference) CPU version at, e.g., 16 nodes (128 cores). This presents the total speedup achieved by LTS, the non-LTS GPU version, and the LTS GPU version.

We also differentiate between simple scaling efficiency, LTS efficiency, and the LTS-scaling efficiency. Listed in each of the performance scaling figures, we list the scaling efficiency of the non-LTS CPU and GPU versions, which simply compares against an ideally scaling code starting at 16 nodes. For the LTS case, LTS scaling efficiency is compared against an ideal LTS code that starts at the speedup predicted by the speedup model (9), and achieves perfect scaling. The CPU version (at, e.g., 16 nodes) typically achieves 100% LTS efficiency, where the GPU version at 16 nodes might only achieve 86% LTS efficiency relative to the predicted speedup using the non-LTS GPU version.

Using the 2.5M element trench model, we evaluated performance from 16 to 128 nodes, with either CPUs or GPUs. The speedup of the LTS version, relative to the original (non-LTS) SPECFEM3D CPU version is highlighted in Fig. 9. Also shown is the ideal LTS scaling curve, which assumes perfect LTS efficiency and scalability. The percentages listed next to LTS-CPU (97%), LTS-GPU (45%), non-LTS CPU (102%), and non-LTS GPU (94%) are the LTS and non-LTS scaling efficiencies relative to their respective ideal scaling curves.

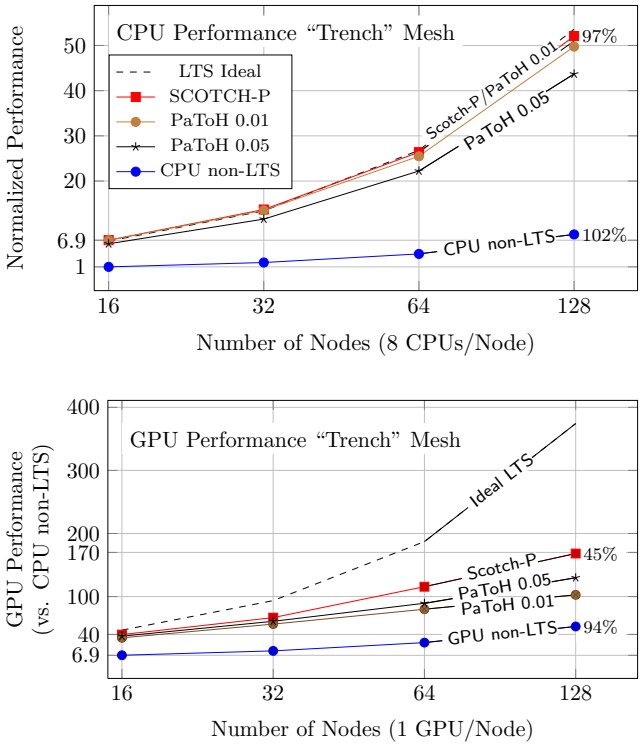

Fig. 9. Performance results of the 2.5M element trench mesh comparing the different LTS partitioning strategies (predicted speedup = 6.7x) using CPUs (top) and GPUs (bottom), relative to the reference (non-LTS) CPU code on 16 nodes. The LTS ideal curve assumes perfect LTS efficiency in addition to perfect scaling. The percentages listed represent the performance fraction relative to the LTS ideal curve and the reference (non-LTS) ideal scaling.

Both PaToH and SCOTCH-P partitioning methods perform very well up to 1024 processors. Both the non-LTS and LTS CPU versions achieve very high scaling efficiency up to at least 128 nodes, which profiling indicates is partially a result of cache performance improving as the partitions grow smaller, which we will analyze in more detail in Sec IV-D. We also compared PaToH performance between the `final_imbal` runtime parameter, where we note that load balance is the correct trade-off for CPU performance.

For the trench example, we also consider GPU performance, as compared to the reference CPU version at 16 nodes. The non-LTS GPU version achieves a speedup of 6.9x over the non-LTS CPU version, and the LTS-GPU with SCOTCH-P starts at 84% LTS efficiency, but is not able to maintain more than 80% efficiency past 32 nodes. Profiling indicates that this scaling inefficiency is mostly due to kernel setup and launch overhead for the very small number of elements in the finer p-levels in each partition. This strong-scaling limitation is an expected limitation of this parallelization method, and is most obvious with the GPU version, where the kernel setup and launch overhead dominates the run time as the number of elements in the smallest levels shrinks.

We further investigate the performance using the "embedding" mesh in Fig. 10. We see similar results to the trench mesh, where SCOTCH-P performs best, followed by the better balanced PaToH. At 16 nodes, SCOTCH-P is able to achieve 95% of the theoretical LTS speedup of 7.9x over the non-LTS CPU version, also at 16 nodes. We again see the super linear scaling attributed to improved cache performance.

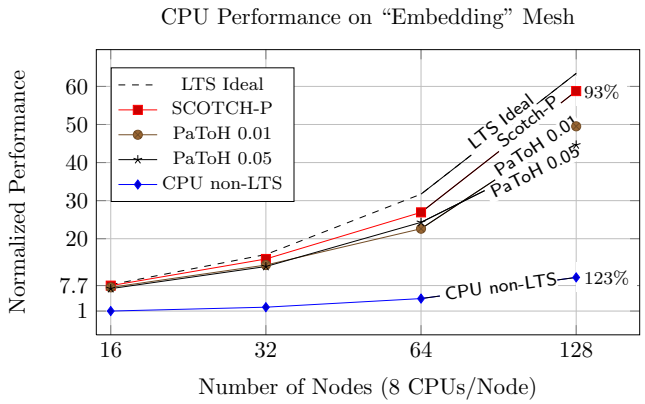

Fig. 10. Performance results on 1.2M element embedding mesh with 7.9x theoretical speedup. We compare SCOTCH-P and PaToH partitioners, including the `final_imbal` PaToH configuration parameter.

We now turn to the "crust" mesh, which is limited to a 1.9x theoretical LTS speedup, due to the large number of small elements along the surface. Seen in Fig. 11, the PaToH and SCOTCH-P partitioning options perform comparably and achieve 96% scaling efficiency at 128 nodes (1024 processors), which is especially important given the limited speedup available for a mesh of this type. Again we see the importance of the stricter load-balancing constraint for PaToH.

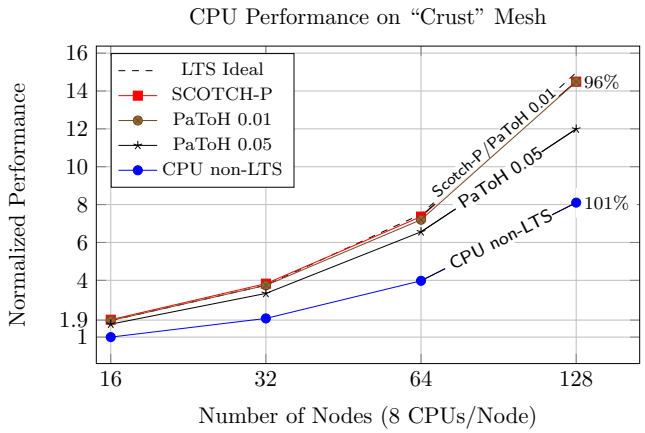

Fig. 11. Performance results on 2.9M element crustal mesh with 1.9x theoretical speedup. We note that the PaToH 0.01 and SCOTCH-P scaling curves are nearly identical.

D. Cache Performance

As noted, the scaling performance of the reference version for the trench mesh exhibits super linear speedup. Although the finite element mesh is quite regular, the layout of the elements and nodal degrees of freedom in memory has never been optimized. Using the Cray performance tool `craypat` to gather a D1+D2 cache utilization metric (hits of L1+L2 data cache), we conducted an experiment using both the reference and LTS versions from 16 to 128 nodes as seen in Fig. 12.

From Fig. 12, we see the expected higher cache use for the reference version coinciding with the super linear scaling performance. We also note that the LTS version of the code

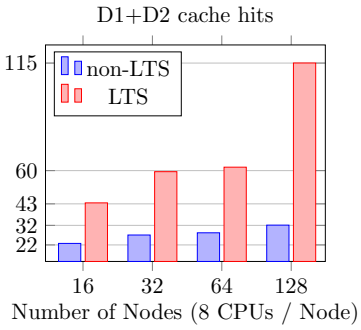

Fig. 12. CPU D1+D2 (L1+L2) level cache hit metric for non-LTS and LTS versions on trench mesh. More hits means greater cache utilization.

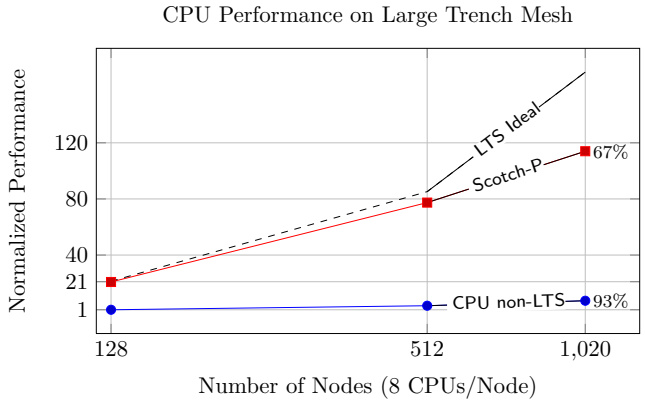

Fig. 13. Performance results on largest 26M mesh for up to 8192 processors across 1024 nodes.

achieves an even greater utilization of cache; the nodal degrees of freedom are grouped by p-level in order to utilize vector operations, which additionally improves cache performance. The LTS algorithm also naturally improves locality because the lowest p-levels contain a small amount of elements (in the ideal case) and require p computations per global Δt, such that many of the memory locations will remain in cache for each $\Delta t/p$ step. We believe that this excellent cache utilization allows the CPU LTS code to overcome the LTS overhead resulting in an efficient scaling. Unfortunately, the GPU version is unable to benefit from these cache advantages, as shown by its lower scaling efficiency.

E. Large Example

The benchmark meshes seen thus far were designed to mirror modestly sized wave-propagation problems across several applications of interest. In the field of computational seismology, there are often several levels of parallelism to exploit, as many real-world applications require many independent source simulations. However, as the problems and computers get bigger, it is important to test large meshes on a large number of processors in order to find bottlenecks in the code that may not have been visible before.

We extended the "trench" example by an order of magnitude to include 26M elements (vs. 2.4M), with an additional refinement layer to increase the theoretical speedup to 21.3x. Figure 13 shows scaling experiments from 128 to 1024 nodes (1024 to 8192 processors, resp.) using the SCOTCH-P partitioner. For this large mesh, the LTS scaling efficiency starts at nearly 100% and remains excellent until 512 nodes (4192 processors), but drops off to 67% at 1024 nodes (8192 processors).

V. Conclusion

In the field of seismic wave propagation, SEMs have seen great success, for their impressive performance and flexibility to utilize user-defined hexahedral meshes. As mentioned, explicit time-stepping schemes enable codes such as SPECFEM3D to achieve impressive scaling efficiency for large problems on large CPU and GPU clusters. However, as we have seen in this paper, the CFL stability criteria can significantly reduce overall performance.

In order to sidestep this CFL bottleneck, we introduced a Newmark LTS method, which has been efficiently implemented in SPECFEM3D, adapted to the continuous nature of the higher-order polynomial basis functions. However, the multi-level nature of our LTS algorithm presents a load-balancing problem for the standard partitioning approach used by the reference code. By formulating this as a multi-constraint partitioning problem, we are able to test several competing partitioning strategies. The multi-constraint hypergraph partitioner provided by the PaToH library produces excellent results and, as noted, can effectively balance the communication and load-balancing constraints, where a similar approach provided by the MeTiS library is unable to compete.

We also propose the relatively simple solution presented as SCOTCH-P, a manual partition of each refinement *p-level*, that provides the best application performance for all of the meshes tested. Overall, we can conclude that LTS can be implemented efficiently for large-scale wave propagation. Additionally, the GPU implementation demonstrates that combining algorithmic and architectural improvements can yield impressive speedups over the original CPU code. Given the move to GPU computing by many supercomputing centers, the effectiveness of the LTS-enabled GPU version should enable application scientists to pursue larger and more complex problems while maintaining or reducing time-to-solution.

Acknowledgment

B. Uçar is supported by France ANR project SOLHAR (ANR13MONU0007). The authors would like to thank the Swiss supercomputing center (CSCS) for their world-class resources and support. D. Peter and M. Rietmann are supported by the Swiss PASC project "A framework for multi-scale seismic modelling and inversion."

References

[1] C. Aykanat, B. B. Cambazoglu, and B. Uçar. Multi-level direct K-way hypergraph partitioning with multiple constraints and fixed vertices. *Journal of Parallel and Distributed Computing*, 68:609–625, 2008.

[2] Claudio Canuto, M Yousuff Hussaini, Alfio Quarteroni, and Thomas A Zang. *Spectral methods: Fundamentals in Single Domains*. Springer, 2006.

[3] Ü. V. Çatalyürek. *Hypergraph Models for Sparse Matrix Partitioning and Reordering*. PhD thesis, Bilkent University, Computer Engineering and Information Science, Nov 1999. Available at http://www.cs.bilkent.edu.tr/tech-reports/1999/ABSTRACTS.1999.html.

[4] Ü. V. Çatalyürek and C. Aykanat. *PaToH: A Multilevel Hypergraph Partitioning Tool, Version 3.0*. Bilkent University, Department of Computer Engineering, Ankara, 06533 Turkey. PaToH is available at http://bmi.osu.edu/~umit/software.htm, 1999.

[5] J. Diaz and M. J. Grote. Energy Conserving Explicit Local Time Stepping for Second-Order Wave Equations. *SIAM J. Sci. Comput.*, 31(3):1985–2014, January 2009.

[6] M. Dumbser, M. Käser, and E. F. Toro. An arbitrary high-order Discontinuous Galerkin method for elastic waves on unstructured meshes - V. Local time stepping and p -adaptivity. *Geophys. J. Internat.*, 171(2):695–717, November 2007.

[7] N. Gödel, S. Schomann, T. Warburton, and M. Clemens. GPU accelerated Adams–Bashforth multirate discontinuous Galerkin FEM simulation of high-frequency electromagnetic fields. *IEEE Trans. Magnetics*, 46(8):2735–2738, 2010.

[8] G. Karypis and V. Kumar. Multilevel algorithms for multi-constraint hypergraph partitioning. Technical Report 99-034, University of Minnesota, Department of Computer Science/Army HPC Research Center, Minneapolis, MN 55455, November 1998.

[9] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on scientific Computing*, 20(1):359–392, 1998.

[10] T. Lengauer. *Combinatorial Algorithms for Integrated Circuit Layout*. Wiley–Teubner, Chichester, U.K., 1990.

[11] Sara Minisini, Elena Zhebel, Alexey Kononov, and Wim A Mulder. Local time stepping with the discontinuous galerkin method for wave propagation in 3d heterogeneous media. *Geophysics*, 78(3):T67–T77, 2013.

[12] F. Pellegrini. *SCOTCH 5.1 User's Guide*. Laboratoire Bordelais de Recherche en Informatique (LaBRI), 2008.

[13] D. Peter, D. Komatitsch, Y. Luo, R. Martin, N. Le Goff, E. Casarotti, P. Le Loher, F. Magnoni, Q. Liu, C. Blitz, T. Nissen-Meyer, P. Basini, and J. Tromp. Forward and adjoint simulations of seismic wave propagation on fully unstructured hexahedral meshes. *Geophys. J. Internat.*, 186(2):721–739, August 2011.

[14] M. Rietmann, P. Messmer, T. Nissen-Meyer, D. Peter, P. Basini, D. Komatitsch, O. Schenk, J. Tromp, L. Boschi, and D. Giardini. Forward and adjoint simulations of seismic wave propagation on emerging large-scale GPU architectures. *Proc. of the Internat. Conf. on High Perf. Comput., Netw., Stor. and Anal.*, November 2012.

[15] M. Rietmann, D. Peter, O. Schenk, and M. Grote. High Performance Newmark Local Time Stepping with Applications in Large Scale Seismic Wave Propagation. *In Preparation*, 2015.